SZAKDOLGOZAT

Korotij Ágnes

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

PROGRAMOZÁSI NYELVEK ONTOLÓGIÁI

Témavezető:
Dr. Juhász István
egyetemi adjunktus

Készítette:
Korotij Ágnes
angol-informatikatanári

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

Jelen dolgozat célja egy ontológiafejlesztési módszertan bemutatása konkrét példán keresztül, mely alkalmas programozási nyelvek ontológiáinak készítésére. A dolgozatban ismertetett esettanulmány az F# programozási nyelv és a .NET CLS szabvány ontológiáin keresztül szemlélteti a módszert.

Az ontológiák készítése során nyilvánosan elérhető forrásokat használtam. A .NET CLS esetén az elsődleges forrásom az ECMA-335 szabvány egy része volt, míg az F# esetén az 1.9.6. Draft Language Specification és az Appress kiadó gondozásában megjelent két elismert szakirodalom, az *Expert F#* és a *Foundations of F#*.

Feltételezésem, hogy a nyelvi specifikációk, beleértve a közismert szabványokat is, a természetes nyelvi leírások jellegzetességéből adódóan inkonzisztensek a szakterületi tudás formalizálásában.

2. Programozási nyelvek ontológiáinak helye az ontológiák csoportosításában

2.1. Az ontológia fogalma

Jelen kontextusban ontológián egy speciális tudásábrázolási technikát értek. Egy közismert definíció szerint „az ontológia egy közös fogalomrendszer formális, explicit specifikációja” ([1], p. 15). Vizsgáljuk meg közelebbről a definíció egyes részeit. A *fogalomrendszer* a jelenség releváns fogalmainak mentális modelljét jelenti. A *közös* arra vonatkozik, hogy egy ontológia nem egyetlen személy szubjektív nézetét tükrözi, hanem az adott területben érintett egyének objektív, konszenzuson alapuló tudását formalizálja. *Expliciten* az egyértelműséget, konzisztenciát értjük. A *formális* pedig arra utal, hogy az ontológiát számítógép által értelmezhető formában kell rögzíteni.

2.2. Az ontológia részei

A tudásreprezentációnak számos változatát ismerjük, csoportosítási alap lehet az ontológia belső szerkezete és témája egyaránt. Ilyen kétdimenziós osztályozásról ír Gómez-Pérez, mely munkának a szerkezeti részre vonatkozó csoportosítását szeretném bemutatni. Gómez-Pérez szerint [6] belső szerkezetük alapján az ontológiákat az alábbi csoportokba sorolhatjuk:

(a) *Korlátolt szókincs*: kifejezések listáját jelenti.
(b) *Glosszárium*: kifejezések és természetes nyelven megadott definícióik.
(c) *Szinonimaszótár*: kifejezések, definícióik és a köztük lévő alapvető szemantikai kapcsolatok (szinonimák és antonímák).
(d) *Informális hierarchiák*: nem szigorú értelemben vett öröklési kapcsolatok, hanem laza viszonyok alapján felállított hierarchia.
(e) *Formális hierarchiák:* más néven taxonómiák; szigorú öröklési („is-a”) kapcsolatok.
(f) *Keretek (frame):* a fogalmaknak osztályok felelnek meg, melyek tulajdonságai öröklődnek.
(g) Ontológiák értékmegszorításokkal
(h) Ontológiák általános logikai megszorításokkal

Ezen a ponton szeretném kiemelni, hogy az ontológiák ilyen jellegű csoportosításának egyes részeivel nem értek egyet. Részben a fent idézett definíció, részben pedig személyes megítélésem alapján a korlátolt szókincs és a glosszárium nem tekinthető ontológiának. Egyik sem formális, és egyik sem ábrázolja a fogalmak közötti kapcsolatokat. A szinonimaszótárok

bár foglalkoznak alapvető kapcsolatokkal, javarészt nem eléggé formálisak. Véleményem szerint az ontológiák a szigorú értelemben vett taxonómiáknál kezdődnek, feltéve hogy öröklődésen kívül más kapcsolatokat is tartalmaz a modell.

Egy tudásreprezentációt akkor tekintünk ontológiának, ha a következő elemeket tartalmazza:

- *Fogalmak*: többféle formában megjelenhetnek, leggyakrabban osztályokkal ábrázolják.
- *Kapcsolatok:* a fogalmak közötti összeköttetéseket ragadja meg, legtöbbször bináris.
- *Megszorítások:* a kapcsolatok jellemzőit kötik feltételekhez.

2.3. Szakterületi ontológiák

Az ontológiákat tárgyuk, témájuk alapján is csoportosíthatjuk, kezdve a legáltalánosabb, magas szintű ontológiáktól (melyek olyan általános fogalmakat próbálnak formalizálni, mint a *tér*, *idő*, *létezés* vagy maga a *fogalom* fogalma) a konkrét tudásterületeket leíró szakterületi ontológiákon át az alkalmazásontológiákig, melyek a meglévő ontológiákat specializálják és használják fel.

A szakterületi ontológiák célja egy konkrét szakterület, például egészségügy, elektronika stb. tudásának a rögzítése. A számítástudományok terén az IEEE felállította a szakterületek taxonómiáját, mely további alkategóriákra osztható 11 fő tudásterületet emel ki. A dolgozat témájának megjelölt programozási nyelvek az IEEE osztályozásában a Szoftver/Szoftverfejlesztés kategóriába esnek. Alcsoportjaik a szabványok, formális leírások és elméletek (beleértve a szintaktikát és a szemantikát egyaránt), nyelvi paradigmák (pl. funkcionális nyelvek, szkript nyelvek, Java – az IEEE ennek a nyelvnek jelentőségénél fogva külön alkategóriát szán), nyelvi szerkezetek és tulajdonságok (pl. adattípusok, keretrendszerek), feldolgozók (pl. fordítók, kódgenerátorok, futtatókörnyezetek) és egy gyűjtő kategória azon területeknek, melyek egyik előbb említett kategóriába sem sorolhatók.

Az IEEE ACM taxonómiájának felső szintű területei (részben kiterjesztve, [7]):

A. Általános szakirodalom
B. Hardver
C. Rendszerszervezés
D. Szoftver/Szoftverfejlesztés
    1. Általános
    2. Programozási technikák

3. Szoftverfejlesztés (i.e. software engineering)
4. Programozási nyelvek
    1) Általános
        a. Szabványok
    2) Formális definíciók és elméletek
    3) Nyelvek osztályozása
    4) Nyelvi eszközök és jellemzők
    5) Feldolgozók
    6) Vegyes
5. Operációs rendszerek

E. Adatok
F. Számításelmélet
G. A számítások matematikája
H. Információtechnológia és információs rendszerek
I. Számítási módszertanok
J. Számítógépes alkalmazások
K. Számítási miliő

Jelen dolgozat a D.4 alkategóriával foglalkozik, azon belül pedig szabványokkal és nyelvi szerkezetekkel.

A programozási nyelvek ontológiáinak szakirodalma igen gyér, mivel a dolgozat írásának idejéig jelentős próbálkozások nem történtek programozási nyelvek ilyen megközelítésben történő formalizálására. A szakterülethez kapcsolódó korábbi kutatásokat Ruiz és Hilera foglalja röviden össze [9]. Cikkükben megemlítik, hogy az MIT-n futó *SIMILE* (Semantic Interoperability of Metadata and Information in unLike Environments – eltérő környezetekből származó metaadatok és információk szemantikai együttműködése) projekt része volt egy „Java ontológia”, ami mindössze az Osztály (Class) és a Csomag (Package) fogalmak közötti strukturális függőségeket írta le. A dolgozat írásának idején ez az ontológia már nem elérhető.

A területhez kapcsolódó említésre érdemes ontológia (szintén Ruiz és Hilera tolmácsolásában, [9], 84. o.) Zimmer és Rauschmayer munkája, akik a forráskód általános ontológiáját készítették el, benne olyan fogalmakkal mint a *Kód (Code)*, *Azonosító (Identifier)* vagy *KódAsszociáció (CodeAssosiation)*. A konkrét programok ezen fogalmakat példányosítják (egy *Java osztály* például a *Kód* fogalom egy példánya).

A programozási nyelvek ontológiáinak készítéséhez Coral Calero és Mario Pattini munkája adta az ötletet, akik az SQL:2003 szabvány ontológiai megközelítését írják le egyik cikkükben [2]. Ontológiájukat UML-ben formalizálták, állításuk szerint az ilyen típusú tudásábrázolás a legjobb kiegészítője egy nagy volumenű szabványnak. Az ontológia készítése során a szerzők több inkonzisztenciát is észrevettek az SQL:2003 szabványban, jóllehet annak csak egy töredékét formalizálták.

3. Ontológiafejlesztési módszertanok

3.1. Problémák

Egy programozási nyelv ontológiájának készítésekor két szempontot kell figyelembe venni. Az egyik az ontológia tárgyát képező programozási nyelv lényege, a másik az ontológia formátuma. Ami a programozási nyelvet illeti, ki kell választani a nyelv modellezendő jellemzőit. Az ontológia formalizmusa kapcsán pedig a legfontosabb feladat egy ontológiafejlesztési módszertan adaptálása vagy kidolgozása, az ábrázolási nyelv kiválasztása és a megfelelő eszköztámogatás biztosítása.

3.2. A programozási nyelvek jellegzetességei

A programozási nyelvek adatszerkezetek és vezérlési szerkezetek együttesével segítik a problémamegoldást. Ezek a szerkezetek programozási nyelvenként változnak, egyes nyelvek el is törlik a kettő közötti megkülönböztetést. Turner és Eden [11] cikkében kifejti a programozás nyelvi ontológiák készítésének problémáit, és bemutatnak néhány szemantikai módszert a nyelvek ábrázolására.

A jelenleg ismert ábrázolásmódok egyike sem nyújt elegáns megoldást a vezérlési szerkezetek modellezésére. A logika-alapú ábrázolásmódok nehézkesek és a szerkezetek növekvő bonyolultságával arányosan csökken kezelhetőségük és karbantarthatóságuk, míg a halmazelméletre épülő megközelítések legfőbb hátránya az, hogy igen mély matematikai tudást feltételeznek. Eddig még egy ontológiának sem sikerült az adattípusok ábrázolásán túl további aspektusokat modelleznie, még a korábban említett SQL:2003 is csak a szabvány típusokra vonatkozó részével foglalkozik.

A fent kifejtett okokból kifolyólag a programozási nyelvek ontológiáinak készítésénél csak a típusrendszereket kíséreltem meg formalizálni, azaz az adatszerkezeteket. A vezérlési szerkezetek modellezése túlmutat jelen dolgozat terjedelmén.

3.3. Az ontológia formalizálása

3.3.1. Az ontológiakészítésnél használt módszer

Figyelemreméltóan sok ontológiafejlesztési folyamat és módszer látott napvilágot az elmúlt néhány évben, melyek zöme nagy hasonlóságot mutat a szoftverfejlesztési folyamatokkal. Ezek egészen részletes leírását olvashatjuk Corcho *et al*. cikkében, mely ontológiafejlesztési elvekről, módszerekről, eszközökről és nyelvekről szól [4]. Az *ontológiafejlesztési folyamat* célja a fejlesztési életciklus (lehetőleg) különálló szakaszainak és

tevékenységeinek felismerése. Egy *ontológiafejlesztési módszer* az ontológia készítésének konkrét lépéseit és technikáit írja le.

Kísérletem tárgya az F# nyelv és ennek részeként a .NET CLS volt, melyek ontológiáinak elkészítése során az alábbi egyszerűsített háromfázisú fejlesztési folyamatot használtam (az egyes lépések beleillenek a szakirodalmak által említett legtöbb folyamatba):

I. Előkészítési szakasz. Ebben a szakaszban választjuk ki a modellezni kívánt szakterületet.

II. Fejlesztési szakasz.

1. *Tervezés*. Ide tartozik az ontológiához felhasznált megbízható forrásanyagok (pl. szabványok, kézikönyvek stb.), illetve a módszer, reprezentációs technikák és eszközök kiválasztása.
2. *Implementáció.* A tudás formalizálása a kiválasztott reprezentáció mentén.
3. *Verifikáció.* A formalizált modell és a forrásanyagok ismételt összevetése, a lehetséges inkonzisztenciák dokumentálása.

III. Utótevékenységek.

1. Az ontológia *kiértékelése*.
2. *Visszatekintés* a folyamatra.
3. *Dokumentáció*.

MEGJEGYZÉS. Bár az utótevékenységek között lett megemlítve, a dokumentáció a teljes folyamatot végig kell hogy kísérje.

Az általam kidolgozott módszer az implementációs, verifikációs és dokumentációs tevékenységek szakaszában alkalmazható. A módszer röviden összefoglalva a következő lépésekből áll (*lásd* 3.1. ábra):

1. *Azonosítsuk az ontológiai szempontból fontos mondatokat*. Olvassuk át a kiválasztott forrásanyagokat, és gyűjtsük ki külön dokumentumba azokat a mondatokat, amelyek szemantikailag értékes információt tartalmaznak. A könnyű visszakereshetőség érdekében minden mondatnál tüntessük fel a pontos forrást (általában elegendő az oldalszámot rögzíteni).
2. *Szervezzük a mondatokat táblázatba.* Készítsünk egy kétoszlopos táblázatot, melynek fejléce „Kifejezés” és „Előfordulás” legyen. Az egyes sorokban egy szakterület-specifikus kifejezés (fogalom) és annak összes előfordulása szerepel. Ha egy mondat több szakterületileg fontos kifejezést is említ, minden egyes fogalomnál fel kell tüntetni. Ez az első ránézésre redundánsnak tűnő

megközelítés a konzisztencia ellenőrzésének egy nagyon fontos segédeszköze. Célszerű ellenőrizni, hogy nem hagytuk-e ki egy kifejezésnek esetleg valamilyen fontos előfordulását. Erre kiválóan alkalmasak a konkordanciák, *lásd* 3.3.3.

3. *Készítsük el, majd pontosítsuk az ontológiát.*

- Társítsunk fogalmakat a kifejezésekhez. Olykor két vagy több kifejezés is ugyanazt a fogalmat takarja, de a változatosság kedvéért felváltva használják őket a szövegben. A modellt meg kell tisztítani az ilyen szinonimáktól. Kevésbé könnyen ismerhető fel a másik szélsőséges eset, amikor egy kifejezés több fogalomra is vonatkozhat, melyek között elsőre nem világos a különbség. A különböző fogalmakat szét kell választani a tudásábrázolásban.

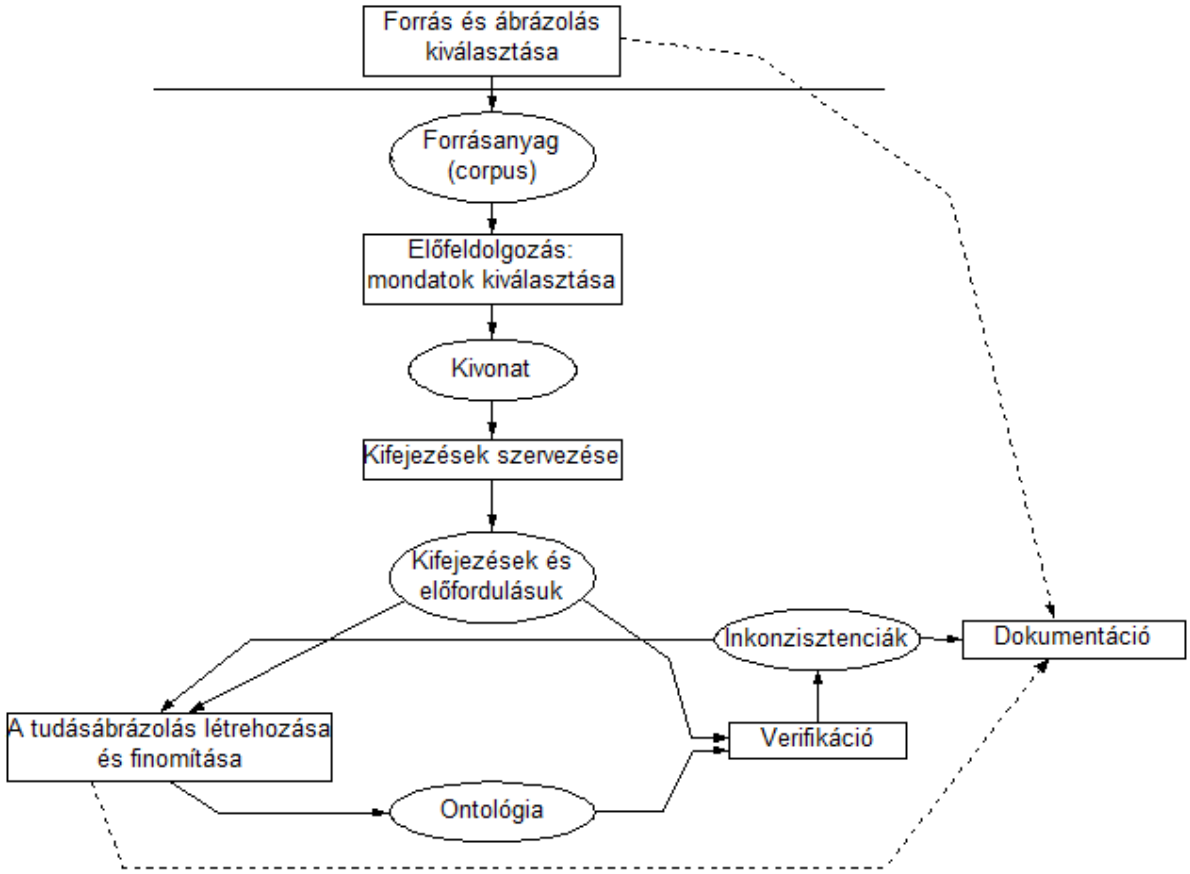
- Azonosítsuk a fogalmak tulajdonságait, attribútumait.

- Tárjuk fel a fogalmakat összekötő kapcsolatokat egyrészt a forrásban található explicit megjelölések, másrészt a fogalmak attribútumainak hasonlósága alapján. A kapcsolatok formalizálásának része a kapcsolat foka (legtöbbször bináris), az érintett fogalmak, a kapcsolat iránya és az egyes végpontokban mért számossága. Igyekezzünk taxonómiákba rendezni a fogalmakat, mert az megkönnyíti kezelésüket, különösen, ha sok fogalommal dolgozunk.

- Ezekkel párhuzamosan figyeljük a forrásszöveg esetleges hiányosságait és inkonzisztenciáit. A modellbe nem vihetünk be inkonzisztenciát. Az inkonzisztenciák feloldásának egyik módja szabványok esetén például a referenciaimplementáció megvizsgálása, amennyiben ilyen létezik.

MEGJEGYZÉS. Ezek a lépések valójában nem választhatók olyan élesen szét, ahogy azt az iménti leírás sugallja. Egy öröklődési kapcsolat például hatással van az ős és a leszármazott fogalom attribútumaira.

4. *Ellenőrizzük a modell konzisztenciáját*. Minden egyes jelentős finomítás után vessük össze a modellt a források kivonatával; vizsgáljuk meg, nem vezettünk-e be inkonzisztenciát. Dokumentáljunk minden ilyen esetet.



3.1. ábra. Ontológiafejlesztési módszer

3.3.2. Ábrázolás

A modell ábrázolásához UML diagramokat, táblázatokat és természetes nyelvi leírásokat használtam.

3.3.2.1. UML

Az UML (Unified Modeling Language) a szoftverfejlesztés területén elterjedt szabványos, általános célú modellező nyelv. Az UML szabványt[1] az Object Management Group[2] (OMG) jegyzi. Az egykor objektumorientált szoftverkomponensek kifejezésére készített nyelv napjainkban széles körben használatos, és kitűnő választás az ontológiák ábrázolásához. Bár több ontológia leíró nyelv is létezik, az UML-t az alábbiak miatt választottam:

- Osztály- (általánosítás és megvalósítás) és példányszintű (asszociáció, aggregáció, kompozíció) kapcsolatok kifejezésére egyaránt alkalmas.
- Kiemelkedő eszköztámogatottsággal bír.

[1] http://www.uml.org
[2] http://www.omg.org

- Szoftverfejlesztők számára értelmezhető. Ez különösen fontos szempont, mivel egy tudásábrázolás csak akkor lehet hasznos, ha megértik azok, akiknek szükségük van rá.
- Intuitíven olvasható és írható.
- Tovább finomítható OCL (Object Constraint Language) megszorításokkal.

3.3.2.2. Táblázatos formátum

Az UML diagramokon nem megjeleníthető információt táblázatokba szervezzük. Kétféle táblázatot használtam: egyet a fogalmak definícióinak összegyűjtésére (egyfajta glosszárium), egy másikat pedig azon kapcsolatok megjelölésére, amelyeket nem ábrázolt a diagram.

3.3.2.3. Természetes nyelv

A természetes nyelv (angol vagy magyar) alkalmas az áttekintések, a definíciók és a megszorítások megfogalmazására. Az általam választott ábrázolásmódban minden ontológiadiagramot egy rövid bevezető előz meg, ami tömören összefoglalja a fogalmakat és a közöttük fennálló kapcsolatokat. A táblázatokat megszorítások követik, azaz további információk, illetve érték-, attribútum- és kapcsolat-megszorítások.

3.3.3. Eszköztámogatás

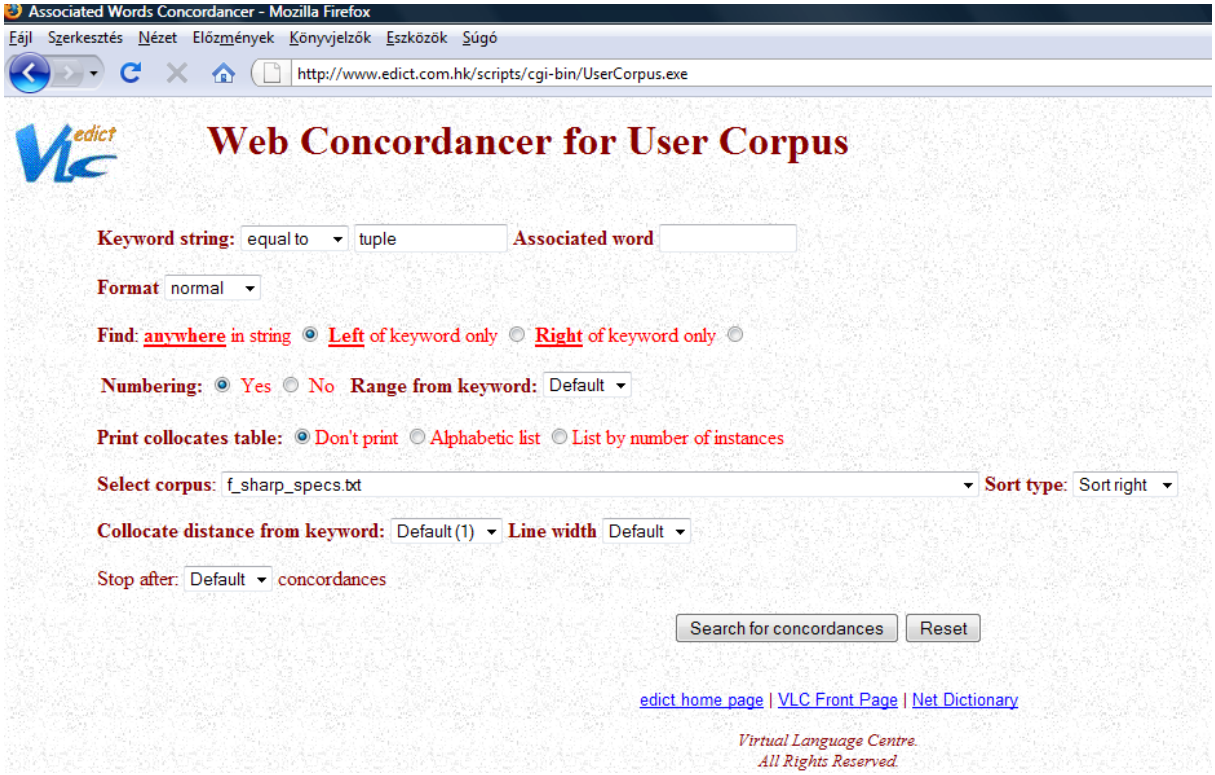
Az eszközök hatással vannak az ontológia elkészítésének idejére és az eredmény minőségére. Az eszközöket két csoportba sorolhatjuk attól függően, hogy függnek-e az ontológia leíró nyelvétől. A legtöbb eszköz a nyelv-függő kategóriába sorolható, ezek részletes leírása megtalálható a [4] cikkben. A dolgozatban bemutatott esettanulmányoknál UML-t használtam, amit számos ingyenes és fizetős eszköz támogat (*lásd* [12] és [13]).

A másik csoportba azok az eszközök tartoznak, amelyek függetlenek az ontológia reprezentációjától. Ezek közül a konkordanciákat szeretném kiemelni, mivel ezeknek lényeges szerep jut a kifejezések előfordulásainak összegyűjtésénél. A tartalmi szempontból releváns állítások kiválasztása és csoportosítása unalmas és sok hibalehetőséget rejt magában. A konkordanciák biztosítják, hogy nem hagyunk ki fontos állításokat. Ez a zömében nyelvészek által kedvelt eszköz alkalmas arra, hogy egy szövegből (ún. korpuszból) kigyűjtsük az összes szót, de ami ennél fontosabb, elsődleges felhasználási módja egy adott kifejezés összes előfordulásának és környezetének listázása. Ha tehát meghatároztuk, hogy melyek azok a kifejezések, amelyeknek az előfordulásait vizsgálni szeretnénk, egy konkordanciával megbizonyosodhatunk arról, hogy valóban nem hagytunk ki egyetlen

előfordulást sem. Ebben a szakaszban egy példán végigvezetve bemutatok néhány ingyenes eszközt, amelynek segítségével konkordanciát készíthetünk. A példánál használt forrás az F# Draft Specification lesz.

A forrásanyagok általában HTML (weboldalak, súgófájlok) vagy PDF (oktatóanyagok, specifikációk, elektronikus könyvek) formátumban érhetők el. A legtöbb konkordanciakészítő program HTML vagy nyers TXT formátummal dolgozik, így a többi formátumban lévő forrásanyagot először át kell konvertálni ezek valamelyikére. Hasznos ingyenes konverterek többek közt az Adobe online konverter[3], a Free PDF to Word Converter[4], és a hasonló funkciókat ellátni képes OpenOffice Writer[5]. A megfelelő formátum elérése után kigyűjthetjük a korpuszból a kívánt kifejezés összes előfordulását. Ehhez kiváló az ingyenes Web Concordancer[6].

Először fel kell töltenünk a korpuszt, majd meg kell adnunk a keresett kulcskifejezést (*lásd* 3.2. ábra). A kilistázott előfordulások (*lásd* 3.3. ábra) bővebb kontextusának megjelenítésére is lehetőségünk van (*lásd* 3.4. ábra).



3.2. ábra. A „tuple” kulcsszó keresése az F# draft specifikációban

[3] http://www.adobe.com/products/acrobat/access_onlinetools.html
[4] http://www.easyfreeware.com/free_pdf_to_word_doc_converter-3933-freeware.html
[5] http://www.openoffice.org/
[6] http://www.edict.com.hk/concordance/

3.3. ábra. A „tuple” kifejezés összes előfordulása a szabványban

3.4. ábra. A kiválasztott kulcsszó tágabb környezete

3.3. A dolgozat további felépítéséről

A dolgozat következő részeiben először bemutatok két ontológia esettanulmányt, majd értékelem az ontológiák készítése során szerzett tapasztalataimat.

Az egyes ontológiákat az alábbi konzisztens felépítés mentén mutatom be:

- Az ontológia tárgyának rövid összefoglalása
- Az al-ontológiák felsorolása
- Minden al-ontológia esetén
  - egy ontológiai diagram
  - fogalmak táblázata
  - kapcsolatok táblázata
  - inkonzisztenciák

4. A .NET CLS ontológiája

4.1. A .NET CLS-ről

A Microsoft .NET keretrendszer az alkalmazásfejlesztést támogató szoftvertechnológia, középpontjában a virtuális gép és a kiterjedt .NET könyvtár. A .NET keretrendszer rugalmassága és ereje a CLI (Common Language Infrastructure – közös nyelvi infrastruktúra), ami lehetővé teszi, hogy a rendszer több különböző programozási nyelvet támogasson. A CLI elsődleges hiteles forrása az ECMA-335 szabvány (továbbiakban: ECMA-335), amely a CLI architektúráját és alapvető fogalmat rögzíti. Az ebben a szakaszban bemutatott definíciók ebből a dokumentumból származnak.

A CLI magja az egységes típusrendszer, a CTS (Commont Type System – közös típusrendszer), melyen a fordítók, a különböző eszközök és maga a CLI osztoznak. Ez a modell definiálja a CLI által a típusok deklarálására, használatára és kezelésére vonatkozó szabályokat.

A CTS gazdag típusrendszere olyan típusokat és műveleteket rögzít, amelyek több programozási nyelvben is előfordulnak. A CTS-t úgy tervezték, hogy programozási nyelvek széles skálájához alkalmazkodjon.

A CLS (Common Language Specification – közös nyelv specifikáció) a nyelvek és a keretrendszerek (azaz az osztálykönyvtárak) fejlesztői közötti egyezmény, a CTS szűk részhalmaza használati konvenciókkal kiegészítve. A programozási nyelvek akkor illeszkednek legjobban a .NET keretrendszerhez, ha a CTS-nek legalább a CLS-ben is előírt részét implementálják. A keretrendszerek pedig akkor használhatók leginkább a .NET keretrendszerből, ha a nyilvánosan közzétett aspektusaik (pl. osztályok, interfészek, metódusok és mezők) megfeleltethetők a CLS-ben lévő típusoknak és a CLS konvencióknak.

4.2. A .NET CLS al-ontológiák

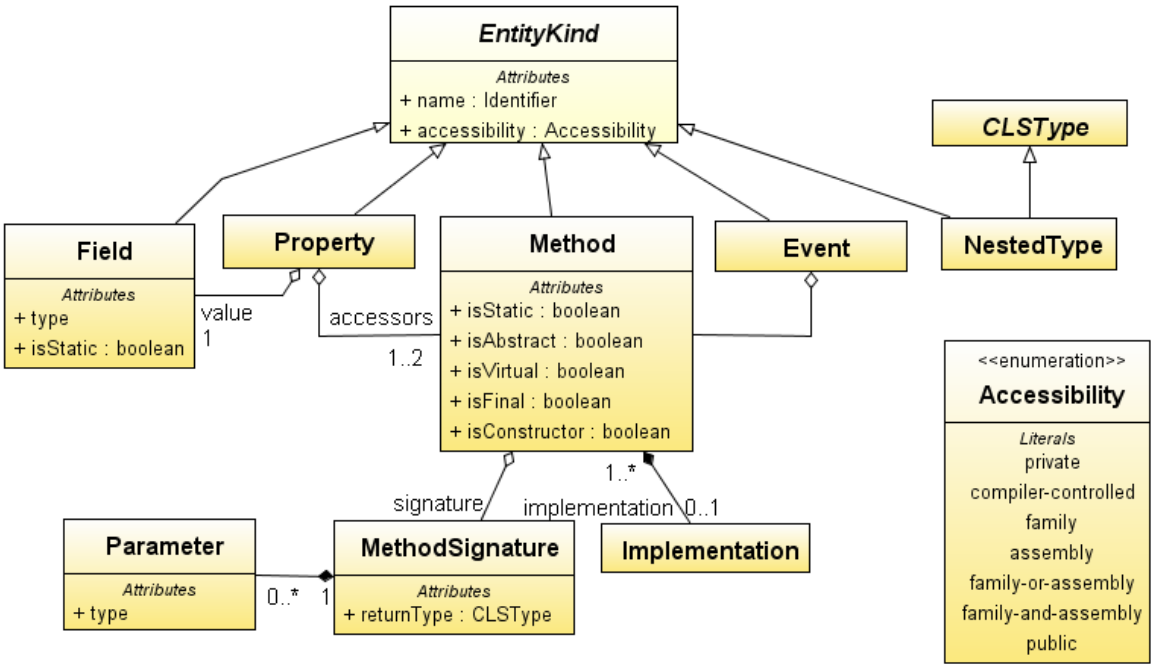
A dolgozat keretein belül csak a CLS-t modelleztem, mivel a .NET könyvtárak csak CLS-kompatibilis típusokat használhatnak, így ez a szűk keresztmetszet. Az ontológia készítése során az alábbi al-ontológiákra bontottam a problémát:

- *Egyedtípusok (Entitiy Kinds)*: Típusok tagjai lehetnek.
- *Egyezmények (Contracts)*: Míg az egyedtípusok a típusdefiníciók konkrét tagjait modellezik, az egyezmények azok absztrakt megfelelőit jelentik. Ez az al-ontológia különösképp az interfész típus modellezéséhez szükséges.
- *CLS típusok (CLS types)*: A CLS-ben definiált típusok, azok csoportosítása és tagjaik.

4.3. Az Egyedtípusok al-ontológia

Egyedtípuson a típusok tagjaiként előforduló entitásokat értjük. Minden egyednek van neve (name) és hozzáférhetősége (accessibility), utóbbi hét előredefiniált érték valamelyike lehet. Ötféle egyedtípust lehet típustag, ezek rendre a mezők (field), metódusok (method), beágyazott típus (nested type), tulajdonságok (property) és események (event). Egy nevesített értéket mezőnek nevezünk. A mezők és metódusok lehetnek statikusak, ami annyit tesz, hogy ilyenkor a típushoz, nem pedig annak konkrét példányaihoz kapcsolódnak. Minden metódusnak van szignatúrája (signature), ami a paraméterek számát, típusát és a metódus visszatérési értékének típusát rögzíti. Ha egy metódust absztrakt módosítóval látunk el, nem tartozik hozzá implementáció. A nem statikus (példányszintű) metódusok lehetnek virtuálisak, a virtuális metódusok pedig lehetnek véglegesek (final). A metódusok egy speciális fajtája a konstruktor. A tulajdonság egy nevesített értéket rejt el egy vagy két hozzáférést biztosító (accessor) metódus mögé. Egy esemény metódusok gyűjteménye. Beágyazott típus bármelyik CLS típus lehet.

Ontológiadiagram



4.1. diagram. Az Egyedtípusok al-ontológia diagram

Táblázatok

Fogalom	Definíció
EntityKind	A kind of entity that can be a type member. Abstract concept.
Field	A kind of entity; represents a named value.
Method	A kind of entity; specifies operations on a type or values of the type.
Property	A kind of entity; specify named values that are accessible via methods that read and write the value.
Event	A kind of entity; specifies named state transitions in which subscribers can register/unregister interest via accessor methods.
NestedType	A kind of entity; Any CLS type.
Implementation	Realizes an operation.
Signature	Specifies the allowable types for all of a method's arguments and for its return value, if any.
Parameter	Typed place holders for arguments.

4.1. táblázat. Az Egyedtípusok al-ontológia fogalmai

Kapcsolatvégpont	Kapcsolatvégpont	Leírás
Property	Field	A property contains a named value, which is a field.
Property	Method	A property may have one or two accessor methods.
Event	Method	An event is a group of accessor methods.
Method	Implementation	A method may have at most one implementation. Abstract methods are not implemented. An implementation may belong to more the one method signature.
Method	Signature	Every method has a signature.
Signature	Parameter	The signature can contain any number of parameters. There are no restrictions on the number of parameters.

4.2. táblázat. Az Egyedtípus al-ontológia kapcsolatai

Megszorítások

(1) Ha egy metódus statikus, akkor nem az értékeken, hanem a típuson végrahajtható műveletet definiált.

(2) Azok a metódusok, melyeknek nincs implementációja, kötelező jelleggel absztrakt metódusok.

(3) Egy objektumtípus nem statikus metódusa megjelölhető virtuálisnak.

(4) A virtuális metódusok lehetnek véglegesek (final).

(5) A statikus mezők nem a típus értékeinek részét képezik, hanem egy, a típushoz kapcsolódó tárterületet hivatkoznak.

(6) 28. CSL szabály: Egy tulajdonságnak lehet lekérdező (getter), beállító (setter) metódusa, vagy ezek mindegyike.

(7) 31. CSL szabály: Egy eseménynek vagy egyszerre lehet `add` és `remove` metódusa, vagy egyik se.

(8) 32. CSL szabály: Az `add` és `remove` metódusok egyetlen paraméterének típusa a `System.Delegate` típus leszármazottja, és az esemény típusát határozza meg.

Inkonzisztenciák

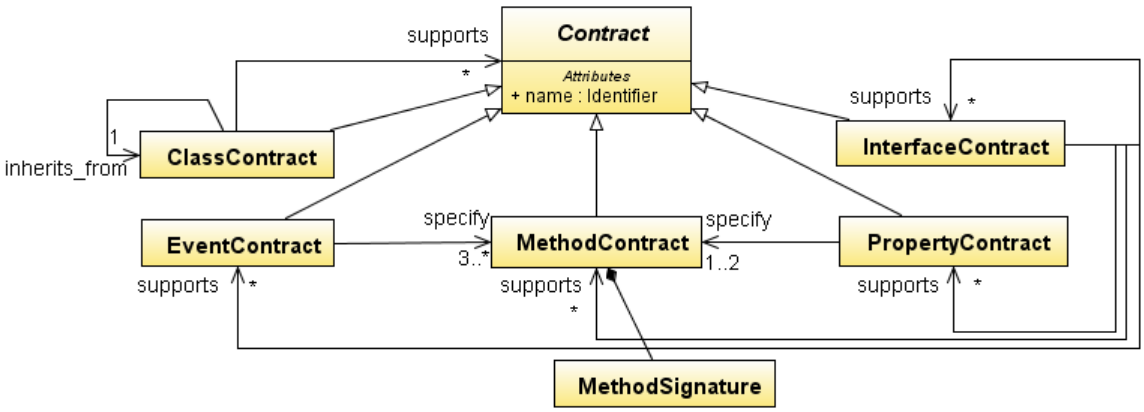
(1) A legtöbb felsorolás csak a mezőket, metódusokat, tulajdonságokat és eseményeket jelöli mint típustag (*lásd* ECMA-335 8.9.6.3., 8.10.), míg a 8.5.2. szakasz a beágyazott típusokat is érvényes típustagnak tartja.

4.4. Az Egyezmények al-ontológia

Az előző al-ontológiával formalizáltuk a típusok lehetséges konkrét tagjait. Az interfészek és absztrakt objektumtípusok ugyanakkor implementáció nélküli egyezményeket rögzítenek, amiből adódik, hogy azok reprezentációjával külön kell foglalkozni.

Az egyezmény (contract) egy nevesített specifikáció, ami implementációra vár. Ötféle egyezményt ismerünk, ezek az osztály, interfész, metódusok, tulajdonság és esemény egyezmények. Az osztály és interfész egyezmények jelezhetik, hogy más egyezményeket is támogatnak (support), de csak osztály egyezmények támogathatnak osztály egyezményeket. Ezt hívjuk más néven öröklődésnek (inheritance). A metódus egyezményt közismertebb nevén szignatúrának nevezzük. Egy tulajdonság legfeljebb két, míg egy esemény egyezmény legfeljebb három metódusszignatúrát deklarálhat.

Ontológiadiagram



4.2. diagram. Az Egyezmények al-ontológia diagram

Táblázatok

Fogalom	Definíció
Contract	Contracts specify the requirements on the implementation of types. Contracts are named.
ClassContract	A class contract specifies the representation of the values of the class type and specifies the other contracts that the class type supports.
InterfaceContract	Interface contracts specify which other contracts the interface supports, e.g. which interfaces, methods, properties and events shall be implemented.
MethodContract	A method contract is a named operation that specifies the contract between the implementation(s) of the method and the callers of the method.
EventContract	An event contract specifies method contracts for all of the operations that shall be implemented by any type that supports the event contract.
PropertyContract	A property contract specifies method contracts for the subset of extensible set of operations for handling a named value, and shall be implemented by any type that supports the property contract.

4.3. táblázat. Az Egyezmények al-ontológia fogalmai

Kapcsolatvégpont	Kapcsolatvégpont	Leírás
ClassContract	Contract	A class contract specifies the other contracts the class type supports. A class contract can support any type of contract.
ClassContract	ClassContract	Every class contract inherits exactly one class contract.
InterfaceContract	InterfaceContract	An interface type can specify support for any number of interface contracts.
InterfaceContract	MethodContract	An interface type can specify support for any number of method contracts.
InterfaceContract	PropertyContract	An interface type can specify support for any number of property contracts.
InterfaceContract	EventContract	An interface type can specify support for any number of event contracts.
MethodContract	MethodSignature	The method contract specifies the contracts that each parameter to the method shall support and the contracts that the return value shall support, if there is a return value. These information are included in the method signature.
PropertyContract	MethodContract	A property contract specifies method contracts for the subset of extensible set of operations for handling a named value. A property contract can specify a getter and/or a setter method contract.
EventContract	MethodContract	An event contract specifies method contracts for at least three standard operations (register interest in an event, revoke interest in an event, fire an event).

4.4. táblázat. Az Egyezmények al-ontológia kapcsolatai

Megszorítások

A diagram minden megszorítást ábrázol, további megszorítások nincsenek.

Inkonzisztenciák

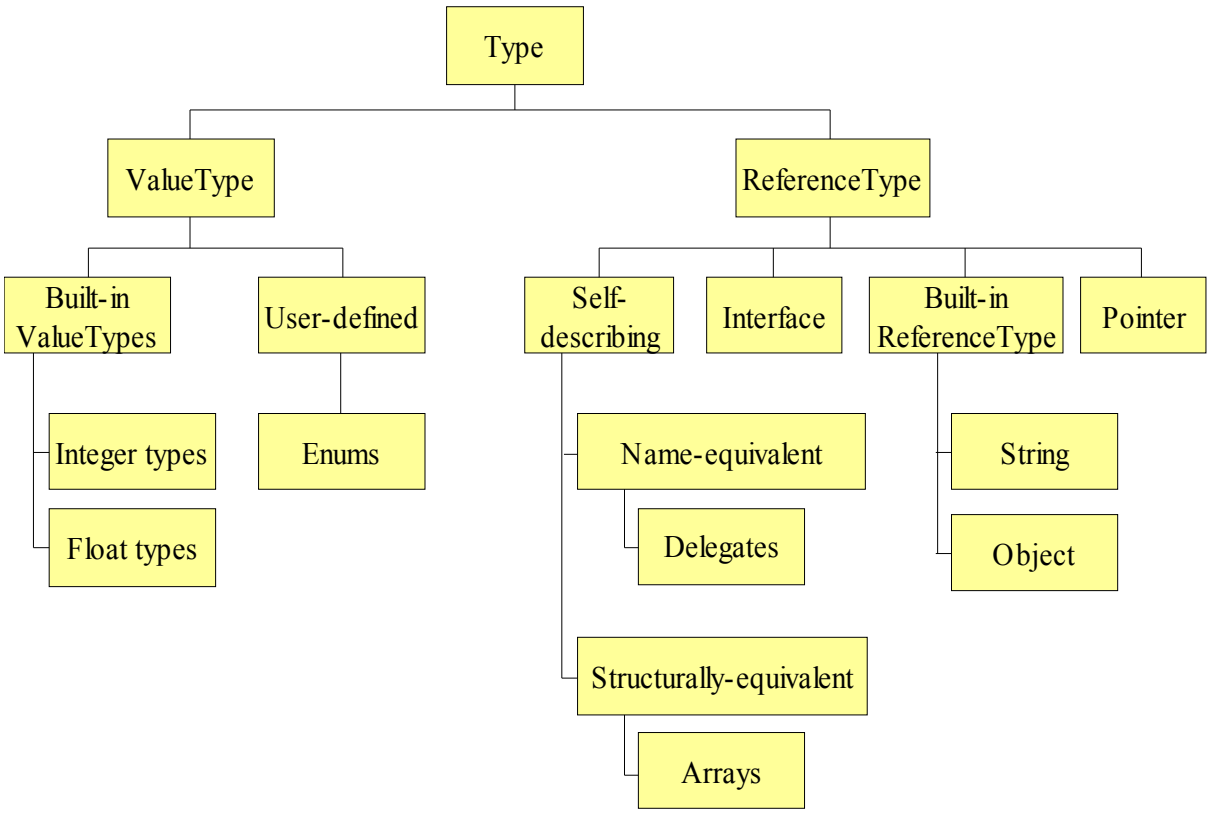
(1) A 8.6. szakasz nem ad megszorításokat arra vonatkozóan, hogy egy osztály egyezmény hány osztály egyezményt támogathat. A 8.9.6.5. szakasz azonban világosan kifejti, hogy egy osztálynak egyetlen őse (base) lehet: „[o]bject type definitions can declare support for one other class contract”. A megszorítást jelöltem a diagramon.

(2) A 8.6. szakasz nem korlátozza a tulajdonságok által deklarálható metódus egyezmények számát. A 28. CLS szabály azonban kimondja, hogy „A property shall have a getter method, a setter method, or both”.

4.5. A CLS típusrendszer al-ontológia

A típusrendszer általános áttekintése

A CLI szabvány az alábbi informális diagramon ábrázolja a típusrendszert (némileg egyszerűsítettem a diagramot azáltal, hogy eltávolítottam a CLS-re nem érvényes részeket):



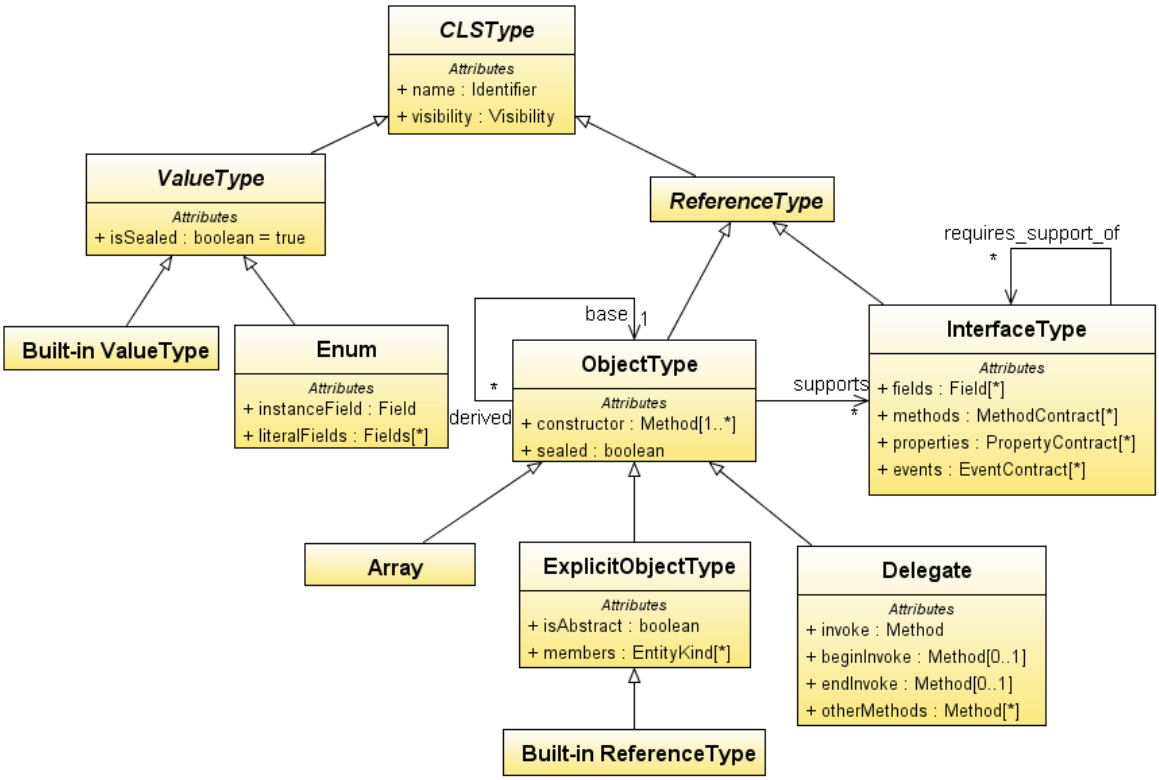
4.1. ábra. A .NET CTS informális diagramjának egyszerűsített nézete

Egy ilyen diagram semmiképpen nem tekinthető ontológiadiagramnak, több okból is. Egyrészt nem pontos az elemek közötti kapcsolatok jelölése: a vonalakat értelmezhetnénk öröklődési/általánosítási kapcsolatként, ezt azonban cáfolja az a tény, hogy a String és az Object referenciatípus példányok, nem pedig azok specializációi. Másrészt a csomópontok elnevezése nem a szabványban használt elnevezéseket követi:

- A mutató (pointer) típust eltávolítottam, mivel az nem CLS-kompatibilis.
- Az „önleíró” („self-describing”) kifejezést csak ez az ábra tartalmazza, a szabvány többi része nem említi. Helyette az „objektumtípus” („object type”) elnevezés használatos.
- Amint azt korábban is említettem, a String és az Object nem leszármazási, hanem példány-kapcsolatban állnak a beépített referenciatípusokkal.
- A beépített referenciatípusok valójában objektumtípusok, így indokolt lenne az önleíró kategória alatt feltüntetni őket.

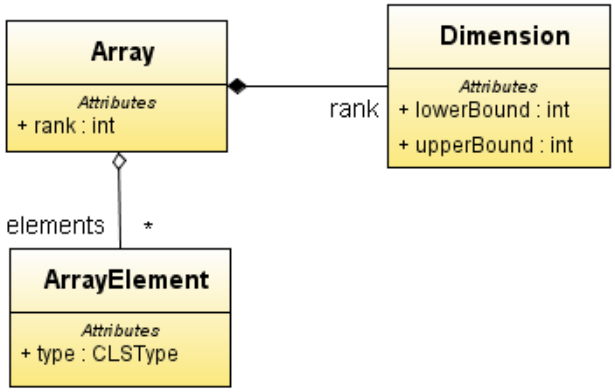
A CLS kétféle típust ismer: értéktípust (value type) és referenciatípust (reference type). Az értéktípusok magukban hordozzák az értéket szemben a referenciatípusokkal, amik egy másik helyen tárolt érték címét hivatkozzák. Az értéktípusokon belül megkülönböztetünk beépített értéktípusokat (built-in value type) és enumokat, vagy más néven enumerációkat (enum[eration]). Az objektumtípus (object type) egy önleíró érték referencia típusa. Az absztrakt osztály (abstract class) olyan objektumtípus, amely az érték részleges definícióját tartalmazza. Az interfész típus (interface type) olyan referenciatípus, amely soha nem definiálja a teljes értéket, ugyanakkor nem objektumtípus. A beépített objektumtípusokat speciálisan kezeli a VES (Virtual Execution System – virtuális futtatórendszer).

Ontológiadiagram



4.3. diagram. CLS típusrendszer al-ontológia diagram

A jobb átláthatóság érdekében a tömböket külön diagramon mutatom be:



4.4. diagram. A tömb típus szerkezete (a CLS típusrendszer része)

Táblázatok

Fogalom	Definíció
CLSType	Types describe values and specify a contract that all values of that type shall support.
ValueType	The values described by a value type are self-contained (each can be understood without reference to other values).
ReferenceType	A value described by a reference type denotes the location of another value.
Built-in ValueType	Given special treatment by the VES.
Enum	Special kind of value type.
InterfaceType	An interface type is a reference type that describes a subset of the operations and none of the representation of a value.
ObjectType	A reference type of a self-describing value.
Delegate	The object-oriented equivalent of function pointers.
ExplicitObjectType	Defined by class definitions, may contain any entity kinds.
Built-in ReferenceType	Given special treatment by the VES. Two instances: System.Object and System.String.
Array	Array objects are defined by the CTS to be a repetition of locations where values of the array element type are stored. An array type shall be defined by specifying the element type of the array, the rank (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array.
Dimension	An array type shall be defined by specifying the element type of the array, the rank (number of dimensions) of the array, and the upper and lower bounds of each dimension of the array.
ArrayElement	A typed location in which a value is stored, reached with the help of indexing expressions.

4.5. táblázat. A CLS típusok al-ontológia fogalmai

Kapcsolatvégpont	Kapcsolatvégpont	Leírás
InterfaceType	InterfaceType	An interface type can require the support of other interface types.
ObjectType	InterfaceType	An object type may support (implement) several interfaces.
ObjectType	ObjectType	An object type inherits from exactly one object type, called its base.
Array	Dimension	An array has at least one dimension.
Array	ArrayElement	An array is a collection of several elements of the same type.

4.6. táblázat. A CLS típusok al-ontológia kapcsolatai

Megszorítások

(1) Ha egy objektumtípus jelzi, hogy egy interfészt támogat, köteles az interfésztípusban deklarált (de nem implementált) összes metódust és egyéb egyezményt implementálni. Egy interfész implementálása tehát az interfész által megkövetelt metódusok implementálását jelenti.

(2) Minden objektumtípus legalább egy konstruktort köteles definiálni, de annak nem feltétlenül kell publikusnak lennie.

(3) Egy interfész (a beágyazott típusok kivételével) csak publikus tagokat deklarálhat:

```
context InterfaceType inv:
        self.methods -> forAll(accessibility = Accessibility.public)
        self.fields -> forAll(accessibility = Accessibility.public)
        self.properties -> forAll(accessibility = Accessibility.public)
        self.events -> forAll(accessibility = Accessibility.public)
```

(4) 19. CLS szabály: A CLS-kompatibilis interfészek nem deklarálhatnak statikus metódusokat, sem mezőket.

```
context InterfaceType inv:
        self.methods -> forAll(isStatic = false)
```

(5) 16. CLS szabály: A tömbök elemeinek típusa CLS-kompatibilis típus lehet. A tömb minden dimenziójának alsó indexe 0.

```
context Array inv:
        self.rank -> forAll(lowerBound = 0)
```

(6) Delegate típus esetén az implementációt a VES biztosítja, nem programozói kód. (MEGJEGYZÉS. Ez az előredefiniált metódusok implementációjára vonatkozik.)

(7) Csak az absztrakt objektumtípus definiálhat olyan metódusegyezményeket, amelyekre a VES nem szolgál implementációval.

(8) 23. CLS szabály: A System.Object CLS-kompatibilis. CLS-kompatibilis osztályok csak CLS-kompatibilis osztályoktól örökölhetnek.

(9) Egy tömb minden elemének egyforma a típusa:

```
context Array inv:
        self.elements -> forAll(e1, e2 : ArrayElement | e1.type = e2.type)
```

(10) Beágyazott típus tagjának, vagy beágyazott típusba ágyazott típusnak a hozzáférhetősége nem lehet megengedőbb az őt definiáló beágyazott típus hozzáférhetőségénél.

(11) A beágyazott típusok láthatósága megegyezik a körülvevő típus láthatóságával.

(12) Egy érték vagy objektumtípus metódusainak neve és szignatúrája egyedi.

(13) Egy interfész minden nem statikus metódusa absztrakt.

(14) Absztrakt metódusdefiníciókat csak az absztrakt jelzővel ellátott objektumtípusok deklarálhatnak.

(15) Objektumtípus nem statikus metódusa lehet virtuális.

(16) Egy interfész definíció minden nem statikus metódusának definíciója virtuális:

```
context InterfaceType inv:
        self.methods -> forAll(isStatic = false implies isVirtual = true)
```

(17) Tulajdonság és esemény definíciókat csak objektumtípusok tartalmazhatnak.

(18) Az érték- és objektumtípusok mezőinek név-típus párja egyedi.

Inkonzisztenciák

(1) Beépített típusok. Az ECMA-335 a beépített típusokat egy táblázatban sorolja fel (*lásd* 8.2.2.), de az értéktípusokat nem választja szét a referenciatípusoktól. Önmagában véve ez nem okoz különösebb problémát, a gond az, hogy a szabvány többi része sem határozza meg, hogy melyik konkrét típus melyik csoportba tartozik.

(2) Hozzáférhetőség vs. láthatóság. A szabvány hét hozzáférési szintet definiál, ezek sorra a private, compiler-controlled, family, assembly, family-or-assembly, family-and-assembly és public. A hozzáférhetőség az egyedtípusok nevének jellemzője. Ebből következik, hogy a típusoknak szintén van hozzáférhetőségük. A láthatóság ezzel szemben azt mondja meg, hogy elérhető-e, azaz látható-e egy típus az assembly-n kívülről, ebben az értelemben pedig két értéket vehet fel, melyek az exported és a not exported. A szabvány több helyen is keveri a láthatóságot és a hozzáférhetőséget, például: “A top-level named type is *exported* if and only if it has public visibility” (8.5.3.1).

(3) Az osztálytípus (class type) kifejezést következetlenül használja a szabvány.

5. Az F# ontológiája

5.1. Bevezetés

5.1.1. Az F#-ról röviden

Az F# a .NET keretrendszerhez készült vegyes paradigmájú programozási nyelv, melyet a .NET elsődleges funkcionális nyelvének tekintenek.

A nyelv funkcionális és imperatív paradigmák vonásait kombinálja, hibrid típusrendszer és vegyes vezérlési szerkezetek jellemzik.

Az F# típusrendszerének jellemzői:

- *Immutabilitás*: Számos F# típus alapértelmezés szerint immutable (megváltoztathatatlan), ami azt jelenti, hogy ha egyszer létrejön egy érték, az a program életciklusa során változatlan marad. Mivel azonban az imperatív programozás megköveteli az értékek változtathatóságát, kompromisszumokkal oldották meg, hogy a .NET CLS típusok (melyek változtathatók) újrafelhasználhatók legyenek az F# programokban.
- *Függvények mint értékek:* A funkcionális paradigma egyik védjegye, hogy a függvények értékek, tehát lehetnek más függvények visszatérési értékei, illetve átadhatók paraméterként. Az érdekes kérdés itt az, hogy hogyan képezhetők le a funkcionális függvények CLS típusokra.
- *Imperatív technikák:* A nyelv lehetőséget biztosít osztálytípusok és interfésztípusok deklarálására, emellett speciális objektum-kifejezéseket is tartalmaz, melyek a .NET könyvtárakkal való együttműködést segítik.

5.1.2. Az ontológia készítéséről

Amint azt a 3. fejezetben is kifejtettem, egy ontológia készítését a forrásanyagok kiválasztása előzi meg. Az F# nyelv esetén elsődleges forrásnak a – draft státuszú – specifikációt tekintettem. Először megpróbáltam elkészíteni a nyelv ontológiáját csak a vázlatos szabványra hivatkozva, de nem sikerült. Egyrészt a dokumentum egy egyszerű taxonómia készítéséhez sem tartalmaz elegendő mennyiségű és minőségű információt. Másrészt a fő hangsúlyt a nyelv szintaktikájára helyezi, miközben elhanyagolja a fontos F# fogalmak definícióját. A kifejezéseket gyakran nem kíséri magyarázat, olykor pedig több jelentésük is ismert.

Mivel az F# draft specifikáció elégtelennek bizonyult, más megbízható forrást kellett keresnem. Választásom két szakkönyvre esett: *Foundations of F#* [8] és *Expert F#* [3]. Ezek a

források azért tekinthetők megbízhatónak, mert szerzőik a nyelv tervezői és olyan személyek, akik szoros kapcsolatban állnak a nyelv fejlesztésével. Az így feldolgozott korpusz mennyisége összesítve 1156 nyomtatott oldal.

5.1.3. Az F# al-ontológiái

A fogalmakat két ontológiába csoportosítottam:

- *F# típusok ontológiája*: Az F# típusrendszerét modellezi két nézet és összesen három diagram formájában.
- *F# típusváltozók megszorításainak ontológiája*: A generikusok kezelését támogató megszorításokat ábrázolja.

5.2. Az F# típusok ontológiája

Az F# a „típus" kifejezést négy különböző, ugyanakkor kapcsolódó értelemben használja. Az ontológiakészítés szempontjából ezen jelentések pontos szétválasztása lényegbevágó. A négy jelentés a következő:

(1) *Típusdefiníció*: Egy adott típus tényleges definíciója a .NET és F# könyvtárakban.
(2) *Szintaktikai típus:* A programszövegben előforduló típusok.
(3) *Statikus típus:* Az ellenőrzés és típuskövetkeztetés során használt típusok, átmenetet képeznek a szintaktikai és futásidejű típusok között.
(4) *Futásidejű típusok:* A típusdefiníciók és statikus típusok információinak futásidejű ábrázolásai.

Az F# típusok elemzésénél végig a statikus típusokra támaszkodom. A dolgozat további részében a „típus" kifejezésen a statikus típusokat értem.

Az ontológia készítése során az egyik legjelentősebb problémám az volt, hogy milyen elvek mentén csoportosítsam az F# típusokat. Felmerült például a funkcionális-imperatív, mutable-immutable, referenciatípus-értéktípus stb. csoportosítás. Mindegyik választással az a probléma, hogy a többi csoportosítás alapját képező tulajdonságokat is fel kell tüntetni a diagramon úgy, hogy az pontos legyen, de ugyanakkor átlátható maradjon. Kezdetben a Pickering által javasolt csoportosítás tűnt vonzónak:

> The type system in F# provides a number of features for defining custom types. All of F#'s type definitions fall into two categories. The first category is types that are *tuples* or *records*. These are a set of types composed to form a composite type (similar to structs in C or classes in C#). The second category is *sum* types, sometimes referred to as *union* types. (p. 42)

Pickering szerint tehát az F# típusok alapvetően két csoportra oszthatók, az első kategóriába a tuple vagy rekord típus kerül, míg a másik csoportba az ún. szum típusok tartoznak, amit gyakran union típusnak is neveznek.

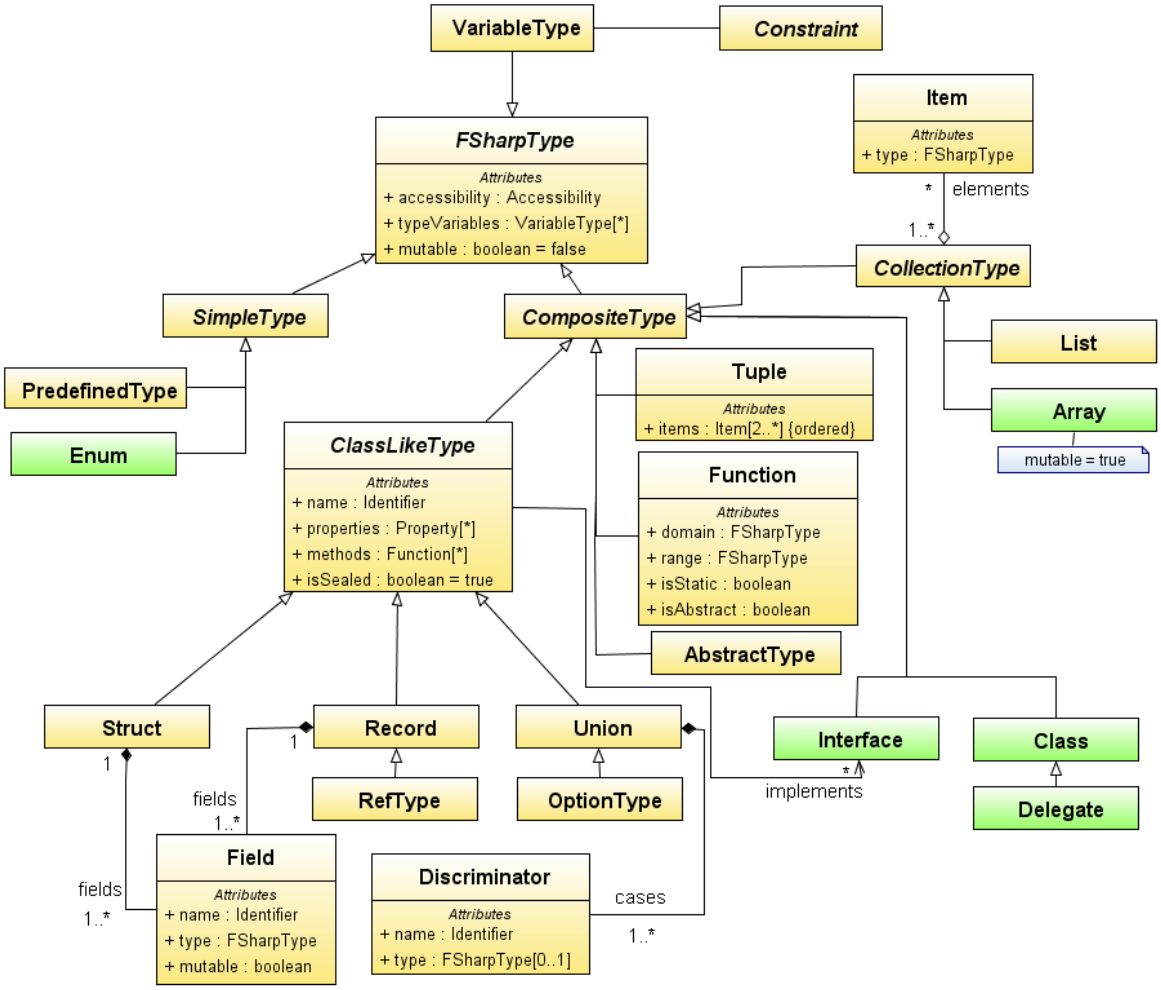
Ennek a megközelítésnek a legnagyobb hátránya, hogy nem ragadja meg a struct, union és record típusok közötti erőteljes hasonlóságot. Az említett típusok mindegyikének lehetnek tagjai (tulajdonságok és metódusok), implementálhatnak interfészeket és zártak (sealed), azaz nem specializálhatók. A közös vonások kiemelése érdekében bevezettem egy közös őst, az osztályszerű típust (class-like type), ami azt a tényt is tükrözi, hogy a struct kivételével ezek a típusok .NET osztály típusra fordítódnak.

Végül úgy döntöttem, az F# típusokat két nézet szerint csoportosítom. Az egyik a típusok szerkezetén, a másik az értéktípus-referenciatípus megkülönböztetésen alapszik.

5.2.1. F# típusok ontológiája, szerkezeti nézet

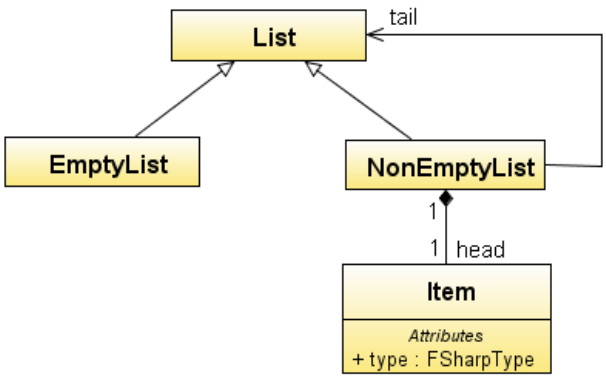
A .NET típusokat zöld háttérrel jelöltem. Ezeket nem fejtem ki a diagramon, részletekért *lásd* a .NET CLS ontológiát. MEGJEGYZÉS. Az F# osztálytípus a CLS objektumtípus megfelelője.

Ontológiadiagram



5.1. diagram. F# típusok ontológiája, szerkezeti nézet

Az áttekinthetőség kedvéért a Lista (List) típust külön diagramon mutatom be:



5.2. diagram. Az F# Lista típusa (az F# típusok ontológiájának részlete)

Táblázatok

Fogalom	Definíció
AbstractType	An incomplete description of a type. May have abstract members.
Array	Arrays are mutable collections. See the CLS Type System Sub-ontology.
Class	See the CLS Type System Sub-ontology, ObjectType.
ClassLikeType	An abstract class to model the similarities between structs, unions and records. All of them may include members (properties and methods), may implement interfaces, but none can serve as a base class.
CollectionType	Represents a collection of items of the same type. The two most popular collections are lists and arrays.
CompositeType	An abstract concept that groups types which are capable to represent complex data.
Constraint	Imposes restrictions on variable types.
Delegate	See the CLS Type System Sub-ontology.
Discriminator	Each alternative of a (discriminated) union is a discriminator. Also called *union case* or *constructor*.
EmptyList	A list that does contains no items.
FSharpType	An abstract concept that represents all the types that the F# language supports.
Function	A reference type whose instances receive values of one type (domain) and produce results of another (possibly same) type (range). Functions may be static or abstract.
Interface	See the CLS Type System Sub-ontology.
Item	Typed part of a composite type.
List	An immutable linked list. It can also be an empty list.
NonEmptyList	A list that can be decomposed into a head item and a tail list.
OptionType	An option type is a union type that can be either None or Some(x), where x is a value of any type.
PredefinedType	Built-in value types, e.g. int.
Record	A class-like reference type that contains named fields. These fields may be mutable. Records may have properties, methods, can implement interfaces and are sealed.
RefType	A ref type is a simple record data structure with a single mutable field.
SimpleType	An abstract concept that groups types which can capture only simple data.
Struct	A class-like value type that contains named fields. Structs may have properties, methods, can implement interfaces and are sealed.

Tuple	Immutable, unnamed type that represents an ordered set of two or more items.
Union	Also called discriminated union, or sum type. A type with one or more discriminated cases. Union types may include members, overrides and interface implementations. They model a final set of choices.
VariableType	Represents a value of any type.

5.1. táblázat. Az F# típusok ontológia fogalmai

Kapcsolatvégpont	Kapcsolatvégpont	Leírás
ClassLikeType	Interface	Class-like types (unions, records and structs) may implement several interfaces.
CollectionType	Item	A collection consists of any number of items.
Record	Field	A record may contain several fields.
Struct	Field	A struct may contain several fields.
Tuple	Item	A tuple groups two or more items into a single structure.
Union	Discriminator	A union is composed of one or more discriminators.
VariableType	Constraint	Variable types may have associated constraints.

5.2. táblázat. Az F# típusok ontológia kapcsolatai

Megszorítások

(1) A discriminated union case nevek nagybetűvel kezdődnek.

(2) Az option típus értéke vagy egy v érték, Some(v), vagy annak hiánya, None:

```
context OptionType inv:
        self.cases -> size() = 2
        self.cases -> exists(c : Discriminator | c.name = 'None')
        self.cases -> exists(c : Discriminator | c.name = 'Some' and c.type -> size() = 1)
```

(3) Az osztályszerű (class-like) típusoknak nem lehet absztrakt metódusa:

```
context ClassLikeType inv:
        self.methods -> forAll(isAbstract = false)
```

(4) A ref típus egyetlen változtatható mezőt tartalmaz, melynek neve „contents”, típusa változó típus:

```
context RefType inv:
        self.fields -> size() = 1
        self.fields -> exists(f : Field | f.name = 'contents' and
                                f.mutable = true and f.type = VariableType)
```

(5) Egy kollekció minden eleme azonos típusú:

```
context CollectionType inv:
        self.elements -> forAll(e1, e2 : Item | e1.type = e2.type)
```

(6) Az F# generikusait részletesebben lásd az 5.2.3. szakaszban.

Inkonzisztenciák

(1) Amint az a bevezetésben is elhangzott, az F# draft specifikáció nem tartalmaz elegendő információt egy hierarchia felépítéséhez. A CLS bizonyos szintű ismerete nélkül lehetetlen az alapján ontológiát készíteni.

(2) A metódus (method) kifejezés határozatlan használata. F# terminológiában a függvény szót részesítik előnyben, típustagoknál viszont a metódus szó gyakoribb. Ezt egyik irodalom sem részletezi.

(3) Az F# draft specifikáció 5.3.1. szakasza a típusrendszer jellemzőit próbálja összefoglalni, azonban elég sikertelenül. Meglátásom szerint a szabványnak ez a legkevésbé konzisztens része, több okból is. Egyrészt egyik felsorolás sem sorolja fel az összes típust, mindegyik csak annak valamelyik részhalmazával foglalkozik. Másrészt az értéktípusok és referenciatípusok közötti megkülönböztetés nem teljes, mivel a felsorolás nem tesz említést az enum, absztrakt osztály, tömb, lista és változó típusokról. Az 5.3.4. szakasz részben orvosolja ezt a problémát azáltal, hogy a korábban kimaradt típusoknak megemlíti az ősosztályát, amiből következtetni lehet jellegére. Ez alapján a tömbök és absztrakt típusok a referenciatípus kategóriába tartoznak, míg az enum értéktípus lesz, .NET terminológia szerint.

(4) Az 5.3.4. szakasz szerint az interfész típus őse a `System.Object`. Ennek ellentmond az ECMA 8.9.11. és 8.6. szakasza: előbbi azt állítja, hogy az objektumtípusokkal ellentétben az interfészek nem alkotnak egyetlen öröklődési hierarchiát („Object types form a single inheritance tree; interface types do not.”), utóbbi szerint pedig interfész típus soha nem lehet osztálytípus, mivel nem tartalmaz teljes értékleírást („An interface type can never fully describe the representation of a value. Therefore an interface type can never support a class contract, and hence can never be a class type or an exact type.”).

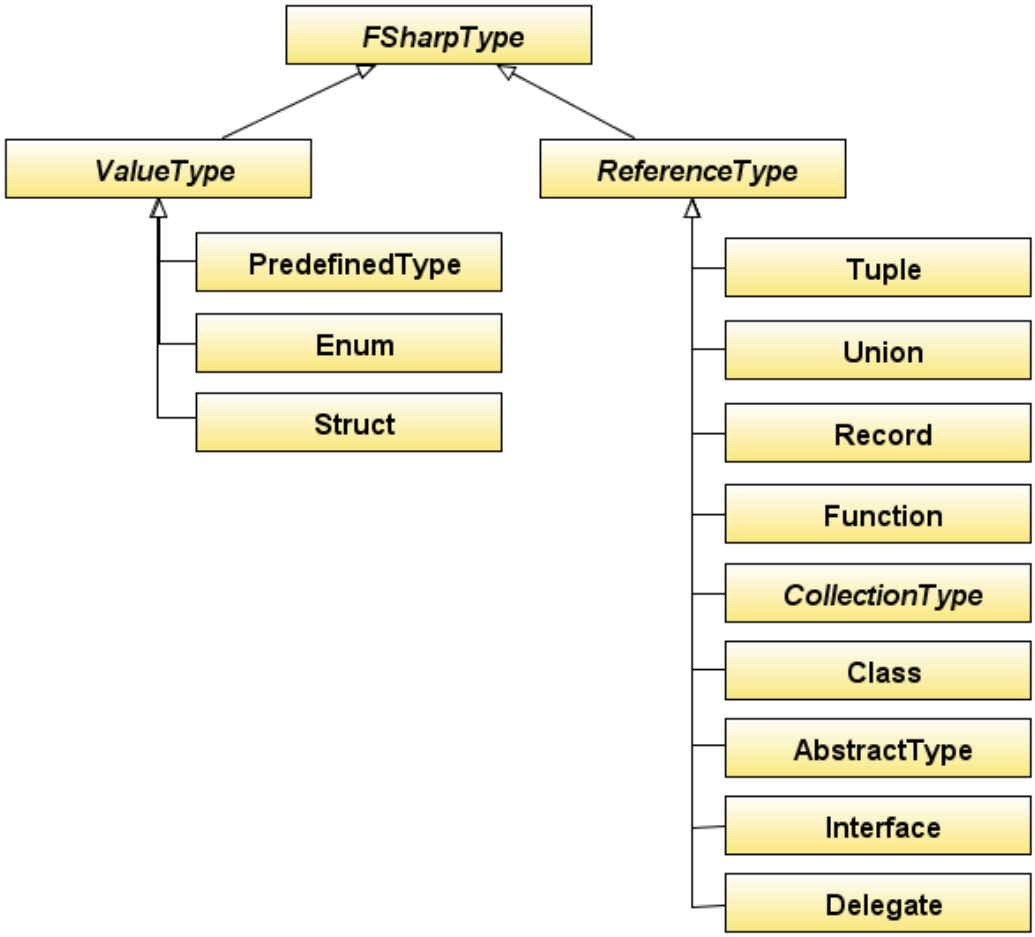
(5) Immutable vs. mutable (változtatható ill. nem változtatható) típusok. A [3] és [8] ellentmondanak azzal kapcsolatban, hogy változtathatónak számít-e a rekord típus. [8] szerint egy rekord frissítése csak a mezők értékét frissíti, nem pedig magát a rekordot (59. o.), eszerint tehát a rekord nem változtatható. [3] ellenben azt állítja, hogy egy rekord változtathatónak tekintendő, ha legalább egyik mezője mutable címkével van ellátva (72. o.). Személyes álláspontom ezen a téren az utóbbi nézethez áll közelebb, vagyis ha a rekord tartalma megváltoztatható, akkor maga a rekord is megváltoztatható. MEGJEGYZÉS. [10] nem foglalkozik a kérdéssel.

5.2.2. F# típusok ontológiája, érték-referencia nézet

Ennél a nézetnél csak a diagramot ismertetem, mivel új fogalmakat nem vezet be.

Az F# három értéktípust ismer, ezek az előredefiniált értéktípusok (pl. `int`, `boolean` stb.), az enumok és a struktúrák. Az összes többi típus a referenciatípusok közé sorolható.

Ontológiadiagram



5.3. diagram. Az F# típusok ontológiája, érték-referencia nézet

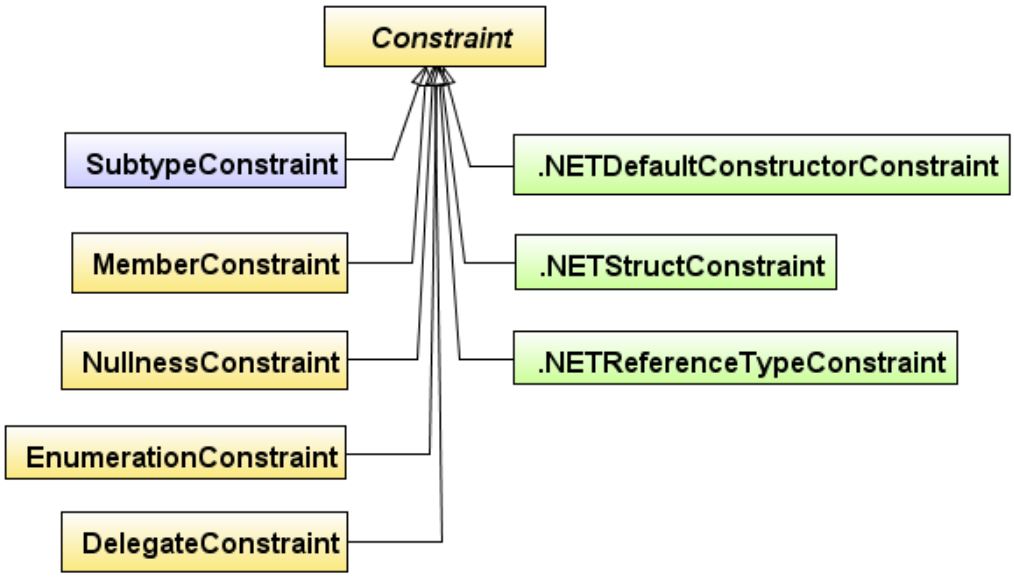
Inkonzisztenciák

Értéktípus vs. struct típus. Az F# draft az `int` típust struct típusnak tekinti: „we say `int` is a struct type because `System.Int32` is a struct type." (F# draft spec., 5.3.2). Az ECMA-335 nem említi a struct kifejezést az értéktípus szinonimájaként, a `System.Int32`-t a beépített értéktípusok között tartja számon. Az ECMA terminológia jellemzően az értéktípus kifejezést használja, míg az F# szakirodalom a struct (struktúra) szinonimával illeti az érték típusú szerkezeteket. Ezzel az a probléma, hogy a struct egyben a C-ből közismert rekord-szerű szerkezet elnevezése is egyben, ez pedig néhol félreértéshez vezethet. Javaslom, hogy a struct kifejezést az utóbbi értelemben használjuk.

5.3. F# változótípus megszorítások al-ontológia (F# generikusok)

A generikusokat F#-ban a típusváltozók jelképezik, melyek típusa kezdetben változó típus. A változótípusokra típusmegszorításokat alkalmazhatunk, melyek szűkítik a típusváltozó lehetséges értékeit. Az 5.4. diagram a típuskövetkeztetésnél használatos megszorításokat ábrázolja. A .NET kompatibilitás érdekében bevezetett megszorításokat zöld színnel jelöltem, a sárga hátterűek ritkán használatosak, az egyetlen tényleges használt megszorítás az altípus megszorítás, amit külön színnel emeltem ki. Az ábra jól mutatja, milyen kompromisszumokat kellett kötni a .NET interoperabilitás érdekében.

Ontológiadiagram



5.4. diagram. F# típusváltozó megszorítások

Táblázatok

Fogalom	Definíció
Constraint	See the F# types ontology.
SubtypeConstraint	A constraint of the form `typar :> type` is an *explicit subtype constraint*.
MemberConstraint	A constraint of the form `(typar or ... or typar) : (member-sig)` is an explicit member constraint. Member constraints are primarily for defining overloaded functions use in the F# library and are used relatively rarely in F# code.
NullnessConstraint	A constraint of the form `typar: null` is an explicit nullness constraint. Nullness constraints are primarily for use during F# type checking and are used relatively rarely in F# code.

EnumerationConstraint	A constraint of the form `typar : enum<underlying-type>` is an *explicit enumeration constraint*. This constraint is met if `typar` is a .NET-compatible enumeration type with constant literal values of type `underlying-type`. This constraint form primarily exists to allow the definition of library functions such as enum. It is rarely used directly in F# programming.
DelegateConstraint	A constraint of the form `typar : delegate<tupled-args-type, return-type>` is an *explicit .NET-framework compatible delegate constraint*. This constraint is met if `typar` is some delegate type D with with declaration `typar D = delegate of object * arg1 * ... * argN` where `tupled-args-type = arg1 * ... * argN`. This constraint form primarily exists to allow the definition of F# library functions. It is rarely used directly in F# programming.
.NETDefaultConstructorConstraint	A constraint of the form `typar : (new : unit -> 'a)` is an explicit .NET default constructor constraint. This constraint is met if `typar` has a parameterless object constructor. This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by .NET. It is rarely used in F# programming.
.NETStructConstraint	A constraint of the form `typar : struct` is an *explicit .NET struct constraint*. This constraint is met if `typar` is a value type, excluding the .NET type `System.Nullable<_>`. This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by .NET. It is rarely used in F# programming.
.NETReferenceTypeConstraint	A constraint of the form `typar : not struct` is an *explicit .NET reference type constraint*. This constraint is met if `type` is a reference type. This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by .NET. It is rarely used in F# programming.

5.3. táblázat. F# típusváltozó megszorítás fogalmai

MEGJEGYZÉS. A megszorítások között csak öröklődési kapcsolat áll fenn, ezeket az ábra tartalmazza, így további táblázatra nincs szükség.

Inkonzisztenciák

A struct kifejezés további félrevezető használata. Az F# draft 5.1.5.5. szakasza által említett .NET Struct Constraint megszorítás akkor áll fenn, ha a típus, amelyikre vonatkozik, .NET értéktípus. Az jelent problémát, hogy a CTS nem definiálja a struct típust. Egyértelmű, hogy mivel a referenciatípussal van szembe állítva, itt az értéktípus értelemben használatos, a struct mégsem érvényes CTS típus.

5.4. Az F# ontológia validálása

Egy ontológia validálása annak ellenőrzése, hogy az ontológiában foglalt fogalmak lefedik-e a szakterület terminológiáját, illetve tükrözik a szakértők véleményét. A .NET CLS

és az F# ontológiák esetében egyértelmű, hogy a szakterület releváns fogalmait tartalmazzák, mivel a területek szabványainak feldolgozásával készültek.

Ami a szakértői validálást illeti, egy ontológia csak akkor érvényes, azaz valid, ha a terület szakértői egyetértenek a benne foglaltakkal. Jelent ontológiát a dolgozatban feltüntetett formájában Gránicz Ádám olvasta át (ezúton is köszönetemet szeretném kinyilvánítani, amiért időt szakított dolgozatomra), aki az ontológia alábbi részeit vitatta:

- Az osztályszerű (class-like) típus létjogosultsága megkérdőjelezhető, mivel az alá sorolt típusok tulajdonságai más referenciatípusokra is illenek. A „sealed" tulajdonság ezzel kapcsolatban nem egyértelműen definiált (keveredik a „lezárt" és az „örökölhetetlen" jelentés). Elképzelhető a class-like típusok áthelyezése közvetlenül a kompozit típusok alá.

- Az ontológia egyes részeinél keveredik a struct szó több jelentése. „A C típusú struktúra (struct in C) egy olyan kompozit típus amely több érteket csomagol egy típusba oly módon, hogy ezen értékek egyenként is megcímezhetők a megfelelő címkével – ez magyarán az altalunk ismert rekord típus, amit C-ben anno struct-nak hívtak. Mivel C-ben fontos volt a rendszerközeli programozás elérhetősége, a struct típus szinonimává vált az „egy adott méretű memóriablokk"-kal, amihez később hozzájárultak a csomagolás szabályait is lehetővé tevő konstrukciók (egymás mellé pakolás vs. bit-padding byte határokig). Ily módon gép-közeli módon megvalósult az igazi érték típus. Mindez a .NET-es nyelveknél, ahol megjelent az OO mint vezérfonal, háttérbe szorult. Ugyanakkor, mivel az eredeti cél optimalizálás szempontjából nem elhanyagolható, a struct tipus .NET (es F#) terminológiában is visszanyerte régi alakját és ily módon egyenlő a value type-pal, és természetesen az F#-os struct típus valóban value type, a rekorddal ill. más osztály alapú típussal ellentétben." Más kifejtésben, a „struct típus 1) magát a struct kulcsszóval ellátott típust, és 2) minden más értéktípust jelenti – egészen egyszerűen azért, mivel minden az utóbbi csoportba tartozó típus modellezhető mint az első".

- Az előredefiniált elnevezés megtévesztő, hiszen vannak nem érték típusú előredefiniált típusok is, bár vitatható, hogy mit nevezünk „előre definiáltnak". A beépített értéktípus szerencsésebb elnevezés lehet.

- A szakértő véleménye szerint egy rekord csak akkor immutabilis, ha „minden mezője 1) érték típusú vagy immutabilis és 2) nem annotált mint mutable". Ez a dolgozatban is említett nézőponthoz áll közelebb, bár annál szigorúbb.

A hivatalos F# szabvány megjelenésével ezeket a javaslatokat be lehet építeni az ontológia következő változatába.

6. Következtetés

A dolgozatban bemutattam a szakterületi ontológiákat, majd ismertettem egy ontológiafejlesztési módszert, amely alkalmas programozási nyelvek ontológiáinak készítésére. Ezt követte két esettanulmány, a .NET CLS és az F# programozási nyelv ontológiái, melyek az ismertetett módszer szerint készültek, továbbá szemléltetik, hogy hogyan nézhet ki egy programozási nyelv ontológiája.

Az esettanulmányok kapcsán említésre érdemes, hogy az ECMA-335 szabvány, bár zömében konzisztens és pontos, redundánsan közelít meg számos témát, aminek következtében nem minden része teljes. Ritkább esetekben a kifejezések használata helytelen, vagy nincs megmagyarázva. Az F# draft specifikáció pedig jelen formájában nem alkalmas arra, hogy egy ontológia kizárólagos forrása legyen.

Az ontológiák készítése során arra a következtetésre jutottam, hogy az ontológiák és a programozási nyelvek összekapcsolása több szempontból is hasznos. Ami a programozási nyelvek specifikációit illeti, az ontológiák kiváló kiegészítői lehetnek a leírásoknak. Először is, egy diagramot és néhány táblázatot könnyebb áttanulmányozni, mint hosszas leírásokat (persze utóbbi nélkülözhetetlen a teljes megértéshez), így például használhatjuk az ontológiákat referenciaként. Másodszor, a specifikációk gyakran nem foglalkoznak külön a kapcsolatok számossági megszorításaival, amire egy ontológia fényt deríthet. A specifikációkban előfordulnak hiányos felsorolások, kimarad egy-két elem – az ontológiadiagramok mindig a teljes képet mutatják. Végezetül pedig az ontológiák segíthetnek csökkenteni a szabványok méretét azáltal, hogy megszüntetik a redundanciát.

Az ontológiák emellett a programozási nyelv fejlesztését is végigkísérhetik. Egy ontológia rávilágít a modell hibáira, ellentmondásaira. Olykor elképzelhető, hogy néhány elem elvételével koherensebb modellt kapunk.

A dolgozat terjedelmi okokból és a téma komplexitásából adódóan csak a típusrendszerek modellezését kísérelte meg. A továbbiakban elképzelhető olyan programozási nyelvi ontológiák készítése, melyek a vezérlési szerkezeteket is lefedik. Ezenfelül további programozási nyelvek modellezése összefüggéseket deríthet fel a nyelvek szerkezetei között, aminek általánosítása egy átfogó programnyelvi ontológia alapját jelentheti.

Ábrák, táblázatok és diagramok jegyzéke

Irodalomjegyzék

[1] Calero, C., Ruiz, F., Pattini, M. (Eds): *Ontologies for Software Engineering and Software Technology*. Berlin: Springer-Verlag. 2006.

[2] Calero, C., Piattini, M.: An ontological approach to the SQL:2003. In *Ontologies for Software Engineering and Technology.* Berlin: Springer-Verlag. Chapter 7. 2006.

[3] Syme, D, Granicz, A., Cisternino, A. *Expert F#.* Apress. 2007.

[4] Corcho, O., Fernández-López, M., Gómez-Pérez, A.: Ontological Engineering: Principles, Methods, Tools and Languages. In *Ontologies for Software Engineering and Technology.* Berlin: Springer-Verlag. Chapter 1. 2006.

[5] ECMA-335 Standard. Common Language Infrastructure. Available at: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[6] Gómez Pérez, A., Fernández López, M. Corcho, O.: *Ontological Engineering*. London: Springer-Verlag. 2004.

[7] IEEE: Top-Level Categories for the ACM Taxonomy (extended version of the ACM Computing Classification System 2002). Available at: www.computer.org/mc/keywords/keywords.htm.

[8] Pickering, Robert. *Foundations of F#.* Apress. 2007.

[9] Ruiz, F., Hilera, J. R.: Using Ontologies in Software Engineering and Software Technology. In: *Ontologies for Software Engineering and Technology*, Berlin: Springer-Verlag. Chapter 2. 2006.

[10] The F# 1.9.6 Draft Language Specification. Available at: http://research.microsoft.com/fsharp/manual/spec2.aspx

[11] Turner, R., Eden, A. H.: Towards a Programming Language Ontology. In: Gordana Dodig-Crnkovic, Susan Stuart (eds.). *Computation, Information, Cognition—The Nexus and the Liminal*. Cambridge, UK: Cambridge Scholars Press. 2007.

[12] UML Tools at the Open Directory Project. Available at: http://www.dmoz.org/Computers/Programming/Methodologies/Modeling_Languages/Unified_Modeling_Language/Tools//

[13] UML Tools at the UML Forum. Available at: http://www.uml-forum.com/tools.htm.